НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №1**

з дисципліни **«**Системне програмування 2**»**

Виконав:

студент 3 курсу гр. ІО-83

Омелянський А. С.

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** Розробка та опрацювання базової структури компілятора

**Мета роботи**: знайомство і опанування основних етапів компілювання, а також створення найпростішого компілятору.

**Варіант** – 20, мова: Python. Система числення значення, що повертається: Decimal, Hex. Типи, що обробляються: int, char.

**Завдання**

1. Ознайомтесь з теоретичними відомостями, додатковою літературою, дайте відповідь на контрольні питання. Це допоможе краще розібратися в роботі.
2. Розробить лексер, що обробляє вхідний файл з розширенням **.py** чи **.с** в залежності від варіанту та повертає список токенів. Ця функція повинна працювати для **всіх** контрольних прикладів лабораторної роботи 1, навіть для “**invalid**”, але, звичайно, обробляти тільки ті типи та системи числення, що дані за варіантом. Програма повинна розбити заданий текстовий рядок на окремі лексеми та побудувати таблицю лексем. Ця таблиця повинна бути відображена у вигляді пар "лексема – тип лексеми", або мати дещо інший вигляд, при умові, що містить ту же інформацію.

Зчитування з файлу рекомендується робити у бінарному представленні (функція **open()** з модифікатором **rb** замість **r** )для коректного сприйняття.

*Від’ємне число не може бути токеном, бо мінус – це окрема унарна операція, що може бути використана з додатнім числом*.

1. Розробить функцію для парсера, що приймає список токенів та повертає **AST**. За бажанням виводити його на екран. Це допоможе краще розуміти структуру компілятора. При цьому можна використовувати будь-який з існуючих способів представлення **AST** в коді. Це може бути клас, тип даних, структура тощо. Функція повинна будувати **AST** для всіх допустимих прикладів (***valid***) та видавати помилку для неприпустимих (***invalid***). Під час приведення типів на екран виводиться позиція приведення. Якщо тип не може бути приведений до int, виводиться помилка. При помилці повинна вказуватись позиція цієї помилки в коді (номер рядка та номер символу).
2. Розробити функцію для генерації асемблерного коду, та записувати згенерований код до файлу.
3. Перевіріти згенерований код шляхом асемблерной вставки або виконананням у середовищі MASM32. Правильний код повинен генеруватись для всіх припустимих (*valid*) прикладів лабораторної роботи та виводити правильний результат.
4. Протестувати програму на декількох своїх контрольних прикладах та відобразити результати тестування у звіті.

**Висновок:**

Я ознайомився і опанував основні етапи компілювання, а також створив найпростіший компілятор мови Python. Розроблені базові структури для лексичного та синтаксичного аналізу, які можуть бути використанні на наступних ітераціях створення більш функціонального компілятору. Розроблено механізм відстеження помилки на всіх фазах компіляції та детальне інформування про неї розробники вхідної програми. Опановано генерацію повноцінного програмного коду віхідної програми на мові assembler x86. Отримані навики можуть бути використанні для розробки більш функціонального компілятору. Результати роботи наведені нижче.

**Результат роботи програми:**

**1. Успішний тест з decimal int**

**test1.py**

def main():

return 12345

**Результат виконання**

C:\prj\io8322\sysprog\lab\1-20-Java-IO-83-Omelyanskyi>PythonCompile test1

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test1.py

[Compiler ] [INFO ] Read 2 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [9 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][RETURN]

[2,12][CONSTINT] -> [12345]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

[Compiler ] [INFO ] Syntax analyzer finished OK.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

{function}:

[STMT\_FUNCTION]

{nameFunction}:

[EXPR\_IDFUNCTION]: "main"

{bodyFunction}:

[STMT\_BLOCK]

{bodyBlock}:

[STMT\_RETURN]

{retValue}:

[EXPR\_CONSTINT]: "12345"

-----------------------------------------------

[Compiler ] [INFO ] Generate destination files

[Compiler ] [INFO ] Result file: test1.asm

[Compiler ] [INFO ] Python Compiler finished OK. [ 25 ms]

===========================================================

=== Execution test1.exe

===========================================================

12345

**test1.asm**

.586

.model flat, stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

NumbToStr PROTO :DWORD,:DWORD

main PROTO

.data

buff db 11 dup(?)

.code

start:

invoke main

invoke NumbToStr, ebx, ADDR buff

invoke StdOut,eax

invoke ExitProcess,0

main PROC

mov ebx,12345

ret

main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD

mov ecx,buffer

mov eax,x

mov ebx,10

add ecx,ebx

LL1:

xor edx,edx

div ebx

add edx,48

mov BYTE PTR [ecx],dl

dec ecx

test eax,eax

jnz LL1

inc ecx

mov eax,ecx

ret

NumbToStr ENDP

end start

**2. Успішний тест з hexical int**

**test2.py**

def main():

return 0x3F5b2

**Результат виконання**

C:\prj\io8322\sysprog\lab\1-20-Java-IO-83-Omelyanskyi>PythonCompile test2

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test2.py

[Compiler ] [INFO ] Read 2 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [9 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][RETURN]

[2,12][CONSTINT] -> [0x3F5b2]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

[Compiler ] [INFO ] Syntax analyzer finished OK.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

{function}:

[STMT\_FUNCTION]

{nameFunction}:

[EXPR\_IDFUNCTION]: "main"

{bodyFunction}:

[STMT\_BLOCK]

{bodyBlock}:

[STMT\_RETURN]

{retValue}:

[EXPR\_CONSTINT]: "259506"

-----------------------------------------------

[Compiler ] [INFO ] Generate destination files

[Compiler ] [INFO ] Result file: test2.asm

[Compiler ] [INFO ] Python Compiler finished OK. [ 32 ms]

===========================================================

=== Execution test2.exe

===========================================================

259506

**test2.asm**

.586

.model flat, stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

NumbToStr PROTO :DWORD,:DWORD

main PROTO

.data

buff db 11 dup(?)

.code

start:

invoke main

invoke NumbToStr, ebx, ADDR buff

invoke StdOut,eax

invoke ExitProcess,0

main PROC

mov ebx,259506

ret

main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD

mov ecx,buffer

mov eax,x

mov ebx,10

add ecx,ebx

LL1:

xor edx,edx

div ebx

add edx,48

mov BYTE PTR [ecx],dl

dec ecx

test eax,eax

jnz LL1

inc ecx

mov eax,ecx

ret

NumbToStr ENDP

end start

**3. Успішний тест з char**

**test3.py**

def main():

return 'T'

**Результат виконання**

C:\prj\io8322\sysprog\lab\1-20-Java-IO-83-Omelyanskyi>PythonCompile test3

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test3.py

[Compiler ] [INFO ] Read 2 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [11 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][RETURN]

[2,12][QUOTE1]

[2,13][CONSTCHAR] -> [T]

[2,14][QUOTE1]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

[Compiler ] [INFO ] Syntax analyzer finished OK.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

{function}:

[STMT\_FUNCTION]

{nameFunction}:

[EXPR\_IDFUNCTION]: "main"

{bodyFunction}:

[STMT\_BLOCK]

{bodyBlock}:

[STMT\_RETURN]

{retValue}:

[EXPR\_CONSTINT]: "84"

-----------------------------------------------

[Compiler ] [INFO ] Generate destination files

[Compiler ] [INFO ] Result file: test3.asm

[Compiler ] [INFO ] Python Compiler finished OK. [ 48 ms]

===========================================================

=== Execution test3.exe

===========================================================

84

**test3.asm**

.586

.model flat, stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

NumbToStr PROTO :DWORD,:DWORD

main PROTO

.data

buff db 11 dup(?)

.code

start:

invoke main

invoke NumbToStr, ebx, ADDR buff

invoke StdOut,eax

invoke ExitProcess,0

main PROC

mov ebx,84

ret

main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD

mov ecx,buffer

mov eax,x

mov ebx,10

add ecx,ebx

LL1:

xor edx,edx

div ebx

add edx,48

mov BYTE PTR [ecx],dl

dec ecx

test eax,eax

jnz LL1

inc ecx

mov eax,ecx

ret

NumbToStr ENDP

end start

**4. Помилка в значенні char**

**test4.py**

def main():

return 'T1'

**Результат виконання**

C:\prj\io8322\sysprog\lab\1-20-Java-IO-83-Omelyanskyi>PythonCompile test4

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test4.py

[Compiler ] [INFO ] Read 2 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][RETURN]

[2,12][QUOTE1]

[2,13][CONSTCHAR] -> [T]

[2,14][NONE.INVALID] -> [1] // Bad closed quote.

-----------------------------------------------

[Lexical ] [ERROR ] test4.py[2,14]->[ return 'T1']

ERR: Bad closed quote.

[Compiler ] [ERROR ] Python Compiler finished ERROR. [ 31 ms]

**5. Помилка в слові return**

**test5.py**

def main():

return1 'T'

**test5.out**

C:\prj\io8322\sysprog\lab\1-20-Java-IO-83-Omelyanskyi>PythonCompile test5

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test5.py

[Compiler ] [INFO ] Read 2 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [11 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][ID] -> [return1]

[2,13][QUOTE1]

[2,14][CONSTCHAR] -> [T]

[2,15][QUOTE1]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

-----------------------------------------------

[Syntax ] [ERROR ] test5.py[2,5]->[ return1 'T']

ERR: Token not keyword "return".

[Compiler ] [ERROR ] Python Compiler finished ERROR. [ 47 ms]

**Лістинг програми:**

**CompileException.java**

package edu.kpi.io8322.sysprog.lab.core;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.lexical.TokenInvalid;

import lombok.Getter;

@Getter

public class CompileException extends Exception {

private int row;

private int col;

public CompileException(int row, int col, String message, Throwable cause) {

super(message, cause);

this.row = row;

this.col = col;

}

public CompileException(TokenInvalid token) {

super(token.getErrMsg(), null);

this.row = token.getRow();

this.col = token.getCol();

}

@Override

public String toString() {

return PythonCompiler.app.getSrcname() +

"[" + row + "," + col + "]->[" + PythonCompiler.app.getSrclines().get(row - 1) + "]" + System.lineSeparator() + String.format("%1$50s %2$s", "ERR: ", getMessage());

}

}

**LexFabric.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import java.util.HashMap;

import java.util.Map;

public class LexFabric {

private Map<LexTypeEnum, LexType> lexTypeMap;

private Map<Character, LexType\_symb> lexTypeSymbMap;

private Map<String, LexType\_keyword> lexTypeKeywordMap;

public LexFabric(LexicalAnalyzer lexicalAnalyzer) {

lexTypeMap = new HashMap<>();

lexTypeMap.put(LexTypeEnum.NONE, new LexType\_none());

lexTypeMap.put(LexTypeEnum.BLOCKINDENT, new LexType\_blockindent());

lexTypeMap.put(LexTypeEnum.BKTB, new LexType\_bktb());

lexTypeMap.put(LexTypeEnum.BKTE, new LexType\_bkte());

lexTypeMap.put(LexTypeEnum.COLON, new LexType\_colon());

lexTypeMap.put(LexTypeEnum.QUOTE1, new LexType\_quote1());

lexTypeMap.put(LexTypeEnum.QUOTE2, new LexType\_quote2());

lexTypeMap.put(LexTypeEnum.DEF, new LexType\_def());

lexTypeMap.put(LexTypeEnum.RETURN, new LexType\_return());

lexTypeMap.put(LexTypeEnum.CONSTINT, new LexType\_constint());

lexTypeMap.put(LexTypeEnum.CONSTCHAR, new LexType\_constchar());

lexTypeMap.put(LexTypeEnum.ID, new LexType\_id());

lexTypeSymbMap = new HashMap<>();

lexTypeSymbMap.put(Character.valueOf('('), (LexType\_symb) getLexType(LexTypeEnum.BKTB));

lexTypeSymbMap.put(Character.valueOf(')'), (LexType\_symb) getLexType(LexTypeEnum.BKTE));

lexTypeSymbMap.put(Character.valueOf(':'), (LexType\_symb) getLexType(LexTypeEnum.COLON));

lexTypeSymbMap.put(Character.valueOf('\''), (LexType\_symb) getLexType(LexTypeEnum.QUOTE1));

lexTypeSymbMap.put(Character.valueOf('\"'), (LexType\_symb) getLexType(LexTypeEnum.QUOTE2));

lexTypeKeywordMap = new HashMap<>();

lexTypeKeywordMap.put("def", (LexType\_keyword) getLexType(LexTypeEnum.DEF));

lexTypeKeywordMap.put("return", (LexType\_keyword) getLexType(LexTypeEnum.RETURN));

}

public LexType getLexType(LexTypeEnum type) {

return lexTypeMap.get(type);

}

public LexType\_symb getLexType\_symb(char value) {

return lexTypeSymbMap.get(Character.valueOf(value));

}

public LexType\_keyword getLexType\_keyword(String value) {

return lexTypeKeywordMap.get(value);

}

}

**LexicalAnalyzer.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.util.ArrayList;

import java.util.List;

@Getter

@Setter

public class LexicalAnalyzer {

private List<String> srclines;

private List<Token> tokenList;

private LexFabric lexFabric;

public LexicalAnalyzer(List<String> srclines) {

this.srclines = srclines;

this.lexFabric = new LexFabric(this);

}

public void exec() throws CompileException {

PythonCompiler.app.logInfo(null, null, "Lexical analyzer starting.");

tokenList = new ArrayList<>();

for (int row = 0; row < srclines.size(); row++) {

String line = srclines.get(row);

int block\_indent = 0;

while (block\_indent < line.length() && line.charAt(block\_indent) == ' ') {

block\_indent++;

}

if (block\_indent > line.length()) continue;

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.BLOCKINDENT), row + 1, 1, line.substring(0, block\_indent)));

int col = block\_indent;

while (col < line.length()) {

if (line.charAt(col) == ' ') {

col++;

continue;

}

LexType lexType = lexFabric.getLexType\_symb(line.charAt(col));

if (lexType != null) {

Token token = new Token(lexType, row + 1, col + 1, null);

tokenList.add(token);

col++;

if (lexType.getType() == LexTypeEnum.QUOTE1 || lexType.getType() == LexTypeEnum.QUOTE2) {

if (col + 1 >= line.length()) {

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, null, "Not a closed quote.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.CONSTCHAR), row + 1, col + 1, line.substring(col, col + 1)));

col++;

LexType lexType2 = lexFabric.getLexType\_symb(line.charAt(col));

if (lexType2 == null || lexType2.getType() != lexType.getType()) {

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, line.substring(col, col + 1), "Bad closed quote.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

tokenList.add(new Token(lexType2, row + 1, col + 1, null));

col++;

}

continue;

}

int next = searchToken(line, col);

if (col == next) {

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, line.substring(col, col + 1), "Bad symbol.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

String tokenValue = line.substring(col, next);

lexType = lexFabric.getLexType\_keyword(tokenValue);

if (lexType != null) {

tokenList.add(new Token(lexType, row + 1, col + 1, null));

col = next;

continue;

}

if (checkConstInt(tokenValue)) {

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.CONSTINT), row + 1, col + 1, tokenValue));

col = next;

continue;

}

if (checkId(tokenValue)) {

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.ID), row + 1, col + 1, tokenValue));

col = next;

continue;

}

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, tokenValue, "Bad identifier.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

}

PythonCompiler.app.logInfo(null, null, "Lexical analyzer finished OK. [" + tokenList.size() + " lexems]");

}

private int searchToken(String line, int start) {

int next = start;

while (next < line.length()) {

if ((line.charAt(next) >= 'A' && line.charAt(next) <= 'Z') ||

(line.charAt(next) >= 'a' && line.charAt(next) <= 'z') ||

(line.charAt(next) >= '0' && line.charAt(next) <= '9') ||

line.charAt(next) == '\_') {

next++;

} else {

return next;

}

}

return next;

}

private boolean checkConstInt(String value) {

if (value.length() > 10) return false;

if (value.startsWith("0x")) {

if(value.length()==2) return false;

for (int i = 2; i < value.length(); i++) {

if (!((value.charAt(i) >= '0' && value.charAt(i) <= '9') ||

(value.charAt(i) >= 'a' && value.charAt(i) <= 'f') ||

(value.charAt(i) >= 'A' && value.charAt(i) <= 'F'))) return false;

}

if (Long.parseLong(value.substring(2), 16) > Integer.MAX\_VALUE) return false;

} else {

for (int i = 0; i < value.length(); i++) {

if (value.charAt(i) < '0' || value.charAt(i) > '9') return false;

}

if (Long.parseLong(value, 10) > Integer.MAX\_VALUE) return false;

}

return true;

}

private boolean checkId(String value) {

if (!((value.charAt(0) >= 'A' && value.charAt(0) <= 'Z') || (value.charAt(0) >= 'a' && value.charAt(0) <= 'z')))

return false;

for (int i = 1; i < value.length(); i++) {

if (!((value.charAt(i) >= 'A' && value.charAt(i) <= 'Z') ||

(value.charAt(i) >= 'a' && value.charAt(i) <= 'z') ||

(value.charAt(i) >= '0' && value.charAt(i) <= '9') ||

value.charAt(i) == '\_'))

return false;

}

return true;

}

public void printTokenList(){

System.out.println("TOKEN LIST:");

System.out.println("-----------------------------------------------");

for(Token token: tokenList){

System.out.println(" "+token.toString());

}

System.out.println("-----------------------------------------------");

}

}

**LexType.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

public abstract class LexType {

private LexicalAnalyzer lexicalAnalyzer;

public abstract LexTypeEnum getType();

}

**LexType\_bktb.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_bktb extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.BKTB;

}

}

**LexType\_bkte.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_bkte extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.BKTE;

}

}

**LexType\_blockindent.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_blockindent extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.BLOCKINDENT;

}

}

**LexType\_colon.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_colon extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.COLON;

}

}

**LexType\_const.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_const extends LexType {

}

**LexType\_constchar.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_constchar extends LexType\_const {

public LexTypeEnum getType(){

return LexTypeEnum.CONSTCHAR;

}

}

**LexType\_constint.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_constint extends LexType\_const {

public LexTypeEnum getType(){

return LexTypeEnum.CONSTINT;

}

}

**LexType\_def.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_def extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.DEF;

}

}

**LexType\_id.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_id extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.ID;

}

}

**LexType\_keyword.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_keyword extends LexType {

}

**LexType\_none.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_none extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.NONE;

}

}

**LexType\_quote1.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_quote1 extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.QUOTE1;

}

}

**LexType\_quote2.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_quote2 extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.QUOTE2;

}

}

**LexType\_return.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_return extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.RETURN;

}

}

**LexType\_symb.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_symb extends LexType {

}

**LexTypeEnum.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public enum LexTypeEnum {

NONE, BLOCKINDENT, SYMB, BKTB, BKTE, COLON, QUOTE1, QUOTE2, KEYWORD, DEF, RETURN, CONST, CONSTINT, CONSTCHAR, ID;

}

**Token.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

@AllArgsConstructor

public class Token {

private LexType lexType;

private int row;

private int col;

private String value;

@Override

public String toString(){

return "["+row+","+col+"]["+lexType.getType()+"]"+(value==null?"":" -> ["+value+"]");

}

public void generateCompileException(String errMsg) throws CompileException {

new TokenInvalid(this, errMsg).throwCompileException();

}

}

**TokenInvalid.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

@Getter

public class TokenInvalid extends Token {

private Token real;

private String errMsg;

public TokenInvalid(Token real, String errMsg) {

super(real.getLexType(), real.getRow(), real.getCol(), real.getValue());

this.real = real;

this.errMsg = errMsg;

}

public TokenInvalid(LexFabric lexFabric, int row, int col, String value, String errMsg) {

super(lexFabric.getLexType(LexTypeEnum.NONE), row, col, value);

this.errMsg = errMsg;

}

public void throwCompileException() throws CompileException {

throw new CompileException(this);

}

@Override

public String toString(){

return "["+getRow()+","+getCol()+"]["+getLexType().getType()+".INVALID]"+(getValue()==null?"":" -> ["+getValue()+"]")+(errMsg==null?"":" // "+errMsg);

}

}

**Expr.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public abstract class Expr extends Node {

public Expr(int row, int col){

super(row, col);

}

public String outGetValue(SyntaxAnalyzer prg) throws CompileException, IOException {

return null;

}

}

**Expr\_Const.java**

package edu.kpi.io8322.sysprog.lab.syntax;

public abstract class Expr\_Const extends Expr {

public Expr\_Const(int row, int col){

super(row, col);

}

}

**Expr\_ConstInt.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Expr\_ConstInt extends Expr\_Const {

private int value;

public Expr\_ConstInt(int row, int col, Token token) throws CompileException {

super(row, col);

switch (token.getLexType().getType()) {

case CONSTINT:

try {

if (token.getValue().startsWith("0x")) {

value = Integer.parseInt(token.getValue().substring(2), 16);

} else {

value = Integer.parseInt(token.getValue(), 10);

}

} catch (Throwable e) {

token.generateCompileException("Bad format number");

}

break;

case CONSTCHAR:

value = token.getValue().charAt(0);

break;

default:

token.generateCompileException("Not integer constant.");

}

}

public NodeType getType(){

return NodeType.EXPR\_CONSTINT;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: \""+value+"\""+System.lineSeparator());

}

public String outGetValue(SyntaxAnalyzer prg) throws CompileException, IOException {

return ""+value;

}

}

**Expr\_Id.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexTypeEnum;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import lombok.Getter;

@Getter

public abstract class Expr\_Id extends Expr {

private String name;

public Expr\_Id(Token token) throws CompileException {

super(token.getRow(), token.getCol());

if(token.getLexType().getType()!=LexTypeEnum.ID) token.generateCompileException("Token not identifier.");

this.name = token.getValue();

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: \""+name+"\""+System.lineSeparator());

}

}

**Expr\_IdFunction.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

public class Expr\_IdFunction extends Expr\_Id {

public Expr\_IdFunction(Token token) throws CompileException {

super(token);

}

public NodeType getType(){

return NodeType.EXPR\_IDFUNCTION;

}

}

**Expr\_IdVar.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

public class Expr\_IdVar extends Expr\_Id {

public Expr\_IdVar(Token token) throws CompileException {

super(token);

}

public NodeType getType(){

return NodeType.EXPR\_IDVAR;

}

}

**Node.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import lombok.Getter;

@Getter

public abstract class Node {

private int row;

private int col;

public Node(int row, int col) {

this.row = row;

this.col = col;

}

public abstract NodeType getType();

public abstract void printTree(StringBuilder buf, String indent);

}

**NodeType.java**

package edu.kpi.io8322.sysprog.lab.syntax;

public enum NodeType {

NONE, EXPR, EXPR\_CONST, EXPR\_CONSTINT, EXPR\_ID, EXPR\_IDFUNCTION, EXPR\_IDVAR,

STMT, STMT\_RETURN, STMT\_BLOCK, STMT\_FUNCTION, STMT\_PROGRAM

}

**Stmt.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public abstract class Stmt extends Node {

public Stmt(int row, int col) {

super(row, col);

}

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

}

}

**Stmt\_Block.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Block extends Stmt {

private Stmt body;

public Stmt\_Block(int row, int col) {

super(row, col);

}

public NodeType getType(){

return NodeType.STMT\_BLOCK;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(body!=null){

buf.append(indent+" {bodyBlock}:"+System.lineSeparator());

body.printTree(buf, " "+indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

if(body!=null){

body.gen(prg, 0, 0);

}

}

}

**Stmt\_Function.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Function extends Stmt {

private Expr\_IdFunction name;

private Stmt\_Block body;

public Stmt\_Function(int row, int col, Expr\_IdFunction name) {

super(row, col);

this.name = name;

}

public NodeType getType() {

return NodeType.STMT\_FUNCTION;

}

public void printTree(StringBuilder buf, String indent) {

buf.append(indent + "[" + getType() + "]" + System.lineSeparator());

buf.append(indent + " {nameFunction}:" + System.lineSeparator());

name.printTree(buf, " " + indent);

if (body != null) {

buf.append(indent + " {bodyFunction}:" + System.lineSeparator());

body.printTree(buf, " " + indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

prg.outWriteln(name.getName()+" PROC");

if(body!=null){

body.gen(prg, 0, 0);

}

prg.outWriteln(name.getName()+" ENDP");

}

}

**Stmt\_Program.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Program extends Stmt {

private Stmt\_Function stmtFunction;

public Stmt\_Program() {

super(1, 1);

}

public NodeType getType(){

return NodeType.STMT\_PROGRAM;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(stmtFunction!=null){

buf.append(indent+" {function}:"+System.lineSeparator());

stmtFunction.printTree(buf, " "+indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

prg.outWriteln(".586");

prg.outWriteln(".model flat, stdcall");

prg.outWriteln("option casemap:none");

prg.outWriteln("include \\masm32\\include\\windows.inc");

prg.outWriteln("include \\masm32\\include\\kernel32.inc");

prg.outWriteln("include \\masm32\\include\\masm32.inc");

prg.outWriteln("includelib \\masm32\\lib\\kernel32.lib");

prg.outWriteln("includelib \\masm32\\lib\\masm32.lib");

prg.outWriteln("NumbToStr PROTO :DWORD,:DWORD");

prg.outWriteln(stmtFunction.getName().getName()+" PROTO");

prg.outWriteln(".data");

prg.outWriteln("\tbuff db 11 dup(?)");

prg.outWriteln(".code");

prg.outWriteln("start:");

prg.outWriteln("\tinvoke main");

prg.outWriteln("\tinvoke NumbToStr, ebx, ADDR buff");

prg.outWriteln("\tinvoke StdOut,eax");

prg.outWriteln("\tinvoke ExitProcess,0");

stmtFunction.gen(prg, 0, 0);

prg.outWriteln("NumbToStr PROC uses ebx x:DWORD,buffer:DWORD");

prg.outWriteln("\tmov ecx,buffer");

prg.outWriteln("\tmov eax,x");

prg.outWriteln("\tmov ebx,10");

prg.outWriteln("\tadd ecx,ebx");

prg.outWriteln("LL1:");

prg.outWriteln("\txor edx,edx");

prg.outWriteln("\tdiv ebx");

prg.outWriteln("\tadd edx,48");

prg.outWriteln("\tmov BYTE PTR [ecx],dl");

prg.outWriteln("\tdec ecx");

prg.outWriteln("\ttest eax,eax");

prg.outWriteln("\tjnz LL1");

prg.outWriteln("\tinc ecx");

prg.outWriteln("\tmov eax,ecx");

prg.outWriteln("\tret");

prg.outWriteln("NumbToStr ENDP");

prg.outWriteln("end start");

}

}

**Stmt\_Return.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Stmt\_Return extends Stmt {

private Stmt\_Function stmtFunction;

private Expr retValue;

public Stmt\_Return(int row, int col, Stmt\_Function stmtFunction, Expr retValue) {

super(row, col);

this.stmtFunction = stmtFunction;

this.retValue = retValue;

}

public NodeType getType(){

return NodeType.STMT\_RETURN;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

buf.append(indent+" {retValue}:"+System.lineSeparator());

retValue.printTree(buf, " "+indent);

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

prg.outWriteln("\tmov ebx,"+retValue.outGetValue(prg));

prg.outWriteln("\tret");

}

}

**SyntaxAnalyzer.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexTypeEnum;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import java.io.IOException;

import java.io.Writer;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class SyntaxAnalyzer {

private List<Token> tokenList;

private Stmt\_Program root;

private int tokenIndexCur;

private Map<Stmt, String> blockIndentMap;

private Stmt\_Function functionCur;

private int labelIndexCur;

private Writer out;

public SyntaxAnalyzer(List<Token> tokenList) {

this.tokenList = tokenList;

}

public Stmt\_Program getRoot(){

return root;

}

public Token tokenCur() throws CompileException {

if (tokenIndexCur < tokenList.size())

return tokenList.get(tokenIndexCur);

throw new CompileException(1, 1, "Attempt to parse past end of file.", null);

}

public Token tokenPeek(int step) {

if (tokenIndexCur + step < tokenList.size())

return tokenList.get(tokenIndexCur + step);

return null;

}

public void tokenNext() {

tokenIndexCur++;

}

public void exec() throws CompileException {

PythonCompiler.app.logInfo(null, null, "Syntax analyzer starting.");

tokenIndexCur = 0;

blockIndentMap = new HashMap<>();

functionCur = null;

root = new Stmt\_Program();

if (tokenCur().getLexType().getType() != LexTypeEnum.BLOCKINDENT)

tokenCur().generateCompileException("Token not block indent.");

blockIndentMap.put(root, tokenCur().getValue());

tokenNext();

root.setStmtFunction(parseFunction());

PythonCompiler.app.logInfo(null, null, "Syntax analyzer finished OK.");

}

public Stmt\_Function parseFunction() throws CompileException {

Token tokenDef = tokenCur();

if (tokenDef.getLexType().getType() != LexTypeEnum.DEF)

tokenDef.generateCompileException("Token not keyword \"def\".");

tokenNext();

Expr\_IdFunction nameFunction = new Expr\_IdFunction(tokenCur());

Stmt\_Function stmtFunction = new Stmt\_Function(tokenDef.getRow(), tokenDef.getCol(), nameFunction);

functionCur = stmtFunction;

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.BKTB) tokenCur().generateCompileException("Not symbol \"(\".");

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.BKTE) tokenCur().generateCompileException("Not symbol \")\".");

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.COLON) tokenCur().generateCompileException("Not symbol \":\".");

tokenNext();

stmtFunction.setBody(parseBlock());

functionCur = null;

return stmtFunction;

}

public Stmt\_Block parseBlock() throws CompileException {

if (tokenCur().getLexType().getType() != LexTypeEnum.BLOCKINDENT)

tokenCur().generateCompileException("Token not block indent.");

Stmt\_Block stmt\_block = new Stmt\_Block(tokenCur().getRow(), tokenCur().getCol());

blockIndentMap.put(stmt\_block, tokenCur().getValue());

tokenNext();

stmt\_block.setBody(parseReturn());

blockIndentMap.remove(stmt\_block);

return stmt\_block;

}

public Stmt\_Return parseReturn() throws CompileException {

Token tokenReturn = tokenCur();

if (tokenReturn.getLexType().getType() != LexTypeEnum.RETURN)

tokenReturn.generateCompileException("Token not keyword \"return\".");

if(functionCur==null) tokenReturn.generateCompileException("Return without function.");

tokenNext();

return new Stmt\_Return(tokenReturn.getRow(),tokenReturn.getCol(), functionCur, parseExpr());

}

public Expr parseExpr() throws CompileException {

Token quoteBegin = null;

if(tokenCur().getLexType().getType()==LexTypeEnum.QUOTE1 || tokenCur().getLexType().getType()==LexTypeEnum.QUOTE2){

quoteBegin = tokenCur();

tokenNext();

}

Expr\_ConstInt expr = new Expr\_ConstInt(tokenCur().getRow(), tokenCur().getCol(), tokenCur());

tokenNext();

if(quoteBegin!=null){

if(tokenCur().getLexType().getType()!=quoteBegin.getLexType().getType()) tokenCur().generateCompileException("Not close quote.");

tokenNext();

}

return expr;

}

public void printTree() {

System.out.println("SYNTAX TREE:");

System.out.println("-----------------------------------------------");

StringBuilder bufPrint = new StringBuilder();

root.printTree(bufPrint, "");

System.out.println(bufPrint);

System.out.println("-----------------------------------------------");

}

public int newLabel() {

labelIndexCur++;

return labelIndexCur;

}

public String strLabel(int num) {

if(num>0) return "L"+num;

return null;

}

public void outWriteLabel(int num) throws IOException {

if (num > 0)

out.write("L" + num + ":" + System.lineSeparator());

}

public void outWriteln(String str) throws IOException {

if (str != null && !str.isEmpty()) out.write(str);

out.write(System.lineSeparator());

}

public void execOut(Writer out) throws CompileException, IOException {

this.out = out;

labelIndexCur = 0;

root.gen(this, 0, 0);

}

}

**PythonCompiler.java**

package edu.kpi.io8322.sysprog.lab;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexicalAnalyzer;

import edu.kpi.io8322.sysprog.lab.syntax.SyntaxAnalyzer;

import lombok.Getter;

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

@Getter

public class PythonCompiler {

private static final String LOG\_FORMAT = "[%1$-10s] [%2$-7s] %3$s";

public static PythonCompiler app;

private String srcname;

private String dstname;

private List<String> srclines;

private LexicalAnalyzer lexicalAnalyzer;

private SyntaxAnalyzer syntaxAnalyzer;

public PythonCompiler(String srcname) {

this.srcname = srcname;

if (srcname.endsWith(".py")) {

dstname = srcname.substring(0, srcname.length() - ".py".length()) + ".asm";

} else {

dstname = srcname + ".asm";

}

}

public int exec() {

logInfo(null, null, "Source file: " + srcname);

srclines = new ArrayList<>();

BufferedReader br = null;

try {

br = new BufferedReader(new FileReader(srcname));

String line = br.readLine();

while (line != null) {

srclines.add(line);

line = br.readLine();

}

br.close();

logInfo(null, null, "Read " + srclines.size() + " rows.");

} catch (Throwable e) {

logError(null, null, "Error read file " + srcname);

if (br != null) {

try {

br.close();

} catch (Throwable e1) {

}

}

return 1;

}

lexicalAnalyzer = new LexicalAnalyzer(srclines);

try {

lexicalAnalyzer.exec();

lexicalAnalyzer.printTokenList();;

} catch(CompileException e){

lexicalAnalyzer.printTokenList();;

logError("Lexical", null, e.toString());

return 1;

}

syntaxAnalyzer = new SyntaxAnalyzer(lexicalAnalyzer.getTokenList());

try {

syntaxAnalyzer.exec();

syntaxAnalyzer.printTree();

} catch(CompileException e){

syntaxAnalyzer.printTree();

logError("Syntax", null, e.toString());

return 1;

}

try {

logInfo(null, null, "Generate destination files");

StringWriter stringWriter = new StringWriter();

BufferedWriter writer = new BufferedWriter(stringWriter);

syntaxAnalyzer.execOut(writer);

writer.close();

String bodyResultFile = new String(stringWriter.getBuffer());

BufferedWriter writerFile = new BufferedWriter(new FileWriter(dstname));

writerFile.write(bodyResultFile);

writerFile.close();

} catch (Throwable e){

logError("Generator", null, e.toString());

return 1;

}

logInfo(null, null, "Result file: " + dstname);

return 0;

}

public void logInfo(String sourceClass, String sourceMethod, String msg) {

if (sourceClass == null) {

System.out.println(String.format(LOG\_FORMAT, "Compiler", "INFO", msg));

} else {

System.out.println(String.format(LOG\_FORMAT, sourceClass, "INFO", msg));

}

}

public void logError(String sourceClass, String sourceMethod, String msg) {

if (sourceClass == null) {

System.out.println(String.format(LOG\_FORMAT, "Compiler", "ERROR", msg));

} else {

System.out.println(String.format(LOG\_FORMAT, sourceClass, "ERROR", msg));

}

}

public static void main(String[] argc) {

if (argc.length < 1) {

System.out.println("Usage: PythonCompiler <source\_file>");

System.out.println();

System.exit(1);

}

PythonCompiler.app = new PythonCompiler(argc[0]);

long timeexec = System.currentTimeMillis();

PythonCompiler.app.logInfo(null, null, "Python compiler");

int exitcode = PythonCompiler.app.exec();

switch (exitcode) {

case 0:

PythonCompiler.app.logInfo(null, null, "Python Compiler finished OK. [ " + (System.currentTimeMillis() - timeexec) + " ms]");

break;

default:

PythonCompiler.app.logError(null, null, "Python Compiler finished ERROR. [ " + (System.currentTimeMillis() - timeexec) + " ms]");

break;

}

System.exit(exitcode);

}

}